UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DESARROLLO DE UN ALGORITMO DE ENJAMBRE PARA ANÁLISIS DE PROBLEMAS BASADOS EN GRAFOS

Diego Moralejo Lorenzo
Tutor: Antonio González Pardo
Ponente: David Camacho Fernández

MAYO 2018

DESARROLLO DE UN ALGORITMO DE ENJAMBRE PARA ANÁLISIS DE PROBLEMAS BASADOS EN GRAFOS

AUTOR: Diego Moralejo Lorenzo
TUTOR: Antonio González Pardo

Applied Intelligence & Data Analysis (AIDA)
Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2018

Resumen (castellano)

En la actualidad, existen una gran cantidad de problemas que se pueden representar usando un grafo, como el problema del coloreado de grafos o la detección de comunidades. Muchos de estos problemas presentan una complejidad computacional elevada, lo que hace necesaria la utilización de algoritmos heurísticos para resolver el problema en un tiempo razonable.

En este Trabajo de Fin de Grado se tratará el problema de la detección de comunidades. Este problema está adquiriendo una especial relevancia en los últimos tiempos, ya que en el mundo actual las redes sociales están en alza, y con ellas los mecanismos de análisis y tratamiento de datos sobre las mismas. Si observamos con atención una red social, y las relaciones entre usuarios que se forman en ella, rápidamente podremos darnos cuenta de que es posible tratarlas como un inmenso grafo en el que los nodos son los usuarios y las aristas las relaciones de amistad (por ejemplo, en Facebook) entre ellos. Para probar los resultados del algoritmo desarrollado sobre un grafo que suponga una demostración de la capacidad real del mismo, hemos elegido un dataset de Facebook, anonimizado y presentado en forma de ego-networks.

El algoritmo de enjambre elegido para resolver el problema de la detección de comunidades sobre el grafo de Facebook es PSO, Particle Swarm Optimization, u Optimización por Enjambre de Partículas, en castellano. PSO permite optimizar un problema a partir de una población de soluciones candidatas, denotadas como "partículas", las cuales se moverán por el espacio de búsqueda según una función de fitness.

Palabras clave (castellano)

Algoritmo de enjambre, Redes sociales, Grafo, Detección de comunidades, Optimización por Enjambre de Partículas, Agrupamiento, Algoritmo heurístico, Topología de grafos, Algoritmo voraz, Entorno discreto.

Abstract (English)

Nowadays, there are a lot of problems that can be solved using a graph representation, such as the graph coloring problem, or community detection problem, among others. Some of these problems have a high level of computational complexity, so the use of heuristic algorithms is required to solve this problems in a reasonable amount of time.

In this Bachelor Thesis, we are going to get into the community detection problem. This is a problem that is getting a lot of attention nowadays, due to the increasing popularity of social media networks, and with them, the tools for social network analisys are rising also. If we take a look at a social network, is easily noticeable that we can treat them as graphs, in wich users are the nodes, and the friendship relationship between them are the edges of the graph. To test the results of our algorithm in a graph that shows its real world posibilities, we have chosen a Facebook dataset, anonymized and presented in form of ego-networks.

The selected swarm algorithm to deal with the problem of community detection on the Facebook graph is PSO, acronym for Particle Swarm Optimization. This algorithm allows us to optimize a problem from a population of candidate solutions, described as “particles”, which will move through the search space, motivated by a fitness function.

Keywords (inglés)

Swarm algorithm, Social Netkorks, Graph, Community Structure Detection, Particle Swarm Optimization, Clustering, Heuristic algorithm, Graph topology, Greedy algorithm, Discrete environment.

Agradecimientos

En primer lugar, agradecer a mi tutor, Antonio González, por su trato amable y atento en todo el proceso de tutelaje del TFG, sus aportaciones, y sus palabras de ánimo.

En segundo lugar, agradecer a David Camacho por prestarse a hacer de ponente.

Por último, agradecer a mis padres, a mis amigos y a mis compañeros de clase, que siempre me han apoyado en todas las decisiones difíciles y en todos los momentos de agobio y estrés que conllevan unos estudios universitarios.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

En el mundo actual, gracias al rápido desarrollo de Internet y de los servicios basados en la Web, y a los avances en la tecnología necesaria para dotarnos de una interacción persona-ordenador natural y cómoda para el usuario final, que ha acercado la Web a la práctica mayoría de hogares del planeta, es de sobra conocida la importancia que han ido ganando a lo largo de los últimos años las redes sociales. Algunas como Facebook o Twitter alcanzan en número de usuarios una porción realmente importante de la población global (Facebook superó durante el último año la escalofriante cifra de dos mil millones de usuarios activos), y están cambiando de forma notable la forma en la que las personas interactúan unas con otras e intercambian información en su día a día, teniendo un gran impacto también en la economía y la forma de elaborar publicidad a nivel mundial.

Con este auge tan notable de este tipo de redes, es indudable que deben emerger también nuevas técnicas para el análisis de datos en redes sociales, puesto que los beneficios de conocer la estructura de estas redes y cómo se distribuye la información a través de ellas es de vital importancia en la sociedad actual. Necesitamos nuevas técnicas que nos ayuden a comprender cómo se han formado estas redes y a predecir fenómenos futuros que acaecerán en ellas, así como detectar usuarios parecidos (con los mismos intereses) mediante las relaciones de amistad que establecen con otros, pues esto es de vital importancia a la hora de orientar sistemas de recomendación o de detectar ciertos comportamientos y matices entre miembros de una misma comunidad. Si podemos establecer comunidades en base a determinadas métricas y podemos separar la red en varios grupos de usuarios similares, se hace notablemente más fácil predecir los gustos y la información que necesitan, que intercambian, y que van a necesitar en el futuro sus usuarios, basándonos en otros usuarios similares.

Los datos que encontramos en las redes sociales se caracterizan por ser grandes volúmenes de datos, con información dinámica y grandes dosis de heterogeneidad e interactividad, lo cual supone un desafío importante para los modelos de datos tradicionales, que son difíciles de aplicar en estos contextos. El análisis de datos de redes sociales es un reto sin precedentes para las técnicas de computación actuales y se está convirtiendo en un área de investigación

que acapara la atención tanto de las ciencias de la información y la computación, la informática y las ciencias sociales, pues afecta de igual manera a todas estas disciplinas.

Desde el punto de vista computacional, supone un desafío de análisis de datos y de resolución de problemas sobre grafos, así como de optimización de procedimientos para lograr manejar cantidades realmente grandes de datos mientras se mantiene un tiempo de ejecución de los algoritmos razonablemente contenido.

Desde el punto de vista social, siempre ha sido objeto de estudio la forma en que las personas se relacionan con su entorno y forman lazos de amistad y apego con otras personas, formando grupos y comunidades de personas con los mismos intereses o actitudes, así como su impacto en la sociedad, en forma de modas y tendencias a seguir.

En esta memoria de Trabajo de Fin de Grado, lo que más nos interesa sin duda es el punto de vista computacional, ya que es el que atañe a la disciplina de estudios que se está desarrollando.

Las ciencias de la computación aplicadas a problemas de redes sociales tienen como objetivo principal hacer uso de los conocimientos propios de este campo para ayudar a la gente a comunicarse y cooperar, y para ayudar a comprender como funciona la sociedad en términos de toma de decisiones.

Para abordar este problema desde una perspectiva más computacional, podemos tomar las redes sociales como si fuesen inmensos grafos en los que los nodos estarían constituidos por los usuarios que forman parte de la red social en cuestión, y las aristas del grafo serían, por tanto, las relaciones entre usuarios mediante el mecanismo que utilice esa determinada red, ya sean relaciones de amistad como en Facebook, con las que formaríamos un grafo no dirigido (puesto que la relación de amistad es siempre recíproca), o relaciones de seguidor y seguido como en Twitter o Instagram, con las que formaríamos un grafo dirigido (ya que el hecho de que un usuario siga a otro no implica que este le siga de vuelta).

El algoritmo que se va a desarrollar se basará en lo propuesto por [1], por lo que se realizará una implementación en Python de un algoritmo avaricioso y discreto basado en Particle

Swarm Optimization para su aplicación en problemas de dominio discreto, como la detección de comunidades en una red social, que se trata realmente de un problema de grafos.

Para las consiguientes pruebas de validación y obtención de resultados, utilizaremos un dataset que representa una porción de la red social *Facebook*, presentada en forma de redes ego. Se ha elegido este dataset en concreto porque con él tenemos la posibilidad de verificar la efectividad del funcionamiento del algoritmo desarrollado, ya que disponemos de la información acerca de cuáles son las comunidades reales presentes en la red, a las que procuraremos que el algoritmo desarrollado se ajuste lo máximo posible.

1.2 Objetivos

El objetivo principal de este Trabajo de Fin de Grado es aplicar un algoritmo de enjambre al problema de la detección de comunidades en redes sociales. Más concretamente, aplicaremos el algoritmo de Particle Swarm Optimization sobre un grafo que representa una muestra de la red social Facebook.

Otro de los objetivos, por tanto, es rediseñar el algoritmo PSO para adaptarlo a nuestro espacio de soluciones, que al tratar con grafos será un espacio discreto, mientras que PSO estaba diseñado originalmente para espacios continuos, y es hasta el momento donde se ha aplicado con más éxito. No obstante, la solución aquí presentada tratará de demostrar que PSO puede ser una buena alternativa para problemas de optimización también sobre espacios discretos, y que es posible aplicarlo con éxito al problema de la detección de comunidades.

Como objetivo general y subyacente aparece el que es sin duda objetivo de cualquier desarrollo sobre análisis de datos de redes sociales, que no es otro sino ayudar a entender el funcionamiento y disposición de las redes sociales, entender cómo se forman, como se propaga y distribuye la información a través de ellas, y ser capaces de predecir fenómenos y comportamientos que afecten a grupos específicos de nodos, así como clasificar los nodos en subgrupos según sus características diferenciales, comprendiendo así cómo se forman las relaciones entre individuos también en el mundo real.

1.3 Organización de la memoria

El resto de esta memoria se distribuye en los siguientes apartados:

- Apartado 2: Estado del arte. En este apartado se describirán los desarrollos actuales en el campo del que trata este Trabajo de Fin de Grado, es decir, se describirán otras aproximaciones a la resolución del problema de detección de comunidades, y se explicará cómo encaja este desarrollo entre ellos, como se complementan y qué ventajas e inconvenientes tiene con respecto a ellos.
- Apartado 3: Diseño. Aquí se explicará en detalle cómo se ha diseñado el algoritmo presentado a lo largo de esta memoria en términos de diseño, así como algunos conceptos teóricos relacionados a tener en cuenta que son necesarios para comprender los detalles de la implementación, que se describirán más adelante.
- Apartado 4: Desarrollo. En este apartado se describirán los detalles de la implementación del algoritmo descrito, así como algunos de los problemas más recurrentes acaecidos durante el proceso de desarrollo y las soluciones que se dieron a los mismos.
- Apartado 5: Pruebas y resultados. En esta sección se pondrá en práctica el algoritmo desarrollado, probando que detecta las comunidades correctas sobre el grafo indicado, y se mostrarán los resultados obtenidos.
- Apartado 6: Conclusiones y trabajo futuro. En este apartado reflexionaremos sobre la funcionalidad desarrollada, argumentando si los resultados son favorables y se han aproximado a lo que esperábamos en un principio, extrayendo las conclusiones oportunas. Asimismo, se considerarán futuros trabajos relacionados de perfeccionamiento o extensión de la funcionalidad aquí descrita.

Al final del presente documento se encontrarán las referencias, el glosario con los términos más técnicos y más lejos del lenguaje natural que puedan generar dudas sobre su significado o acepción al lector, y los diferentes anexos, a los que se hará referencia en los apartados oportunos.

2 Estado del arte

2.1 Detección de comunidades

Las redes sociales están basadas en comportamientos sociales. Como hemos expuesto durante el apartado anterior, en teoría de grafos, una red social puede expresarse como un grafo compuesto por nodos y aristas. El objetivo de la detección de comunidades es separar la red en subconjuntos más pequeños de nodos, entre los cuales se maximizan las similitudes entre los miembros del mismo subconjunto y miembros de diferentes subconjuntos son muy diferentes. Estos subconjuntos se denominan habitualmente comunidades; aunque también suelen denominarse clústeres o círculos.

La detección de comunidades supone uno de los pilares centrales del análisis de la estructura de las redes sociales. Este problema, también denominado en ocasiones como "clústering de redes sociales", se basa en descubrir la organización de una red, es decir, se basa en la organización de sus vértices (o nodos) en distintas comunidades (o clústeres) que representan subgrafos de la red original. El clústering de grafos consiste en dilucidar qué grupos de vértices bien diferenciados pueden observarse dentro de un mismo grafo. Estos subgrupos, o clústeres, se diferencian normalmente por tener sus nodos alguna característica común que no comparten con otros nodos de la red, o que es más notable entre ellos que entre uno de ellos y los demás nodos de la red fuera del subgrupo. Estas diferencias pueden estar basadas, por ejemplo, en el número de aristas entre nodos del mismo grupo con respecto a aristas entre los nodos pertenecientes al grupo y los que no lo son. Si un grupo de nodos es una comunidad, será probable que el número de aristas "intracomunitarias" (es decir, entre los nodos de una misma comunidad) sea mucho mayor que el número de aristas "intercomunitarias" (es decir, entre nodos de distintas comunidades).

Tal y como sea expuesto en el apartado 1.1, en la motivación, podemos considerar una red social como un grafo en el que se representan los nodos como los usuarios de la red y las aristas como las relaciones entre ellos, de forma que podemos reducir todos los problemas que queramos resolver sobre la red social a un problema de grafos.

Los problemas de grafos siempre han sido una constante en el mundo de la computación y de la informática, pues son útiles para expresar gran cantidad de problemas computacionales tales como máquinas de estados, árboles de decisión, algoritmos de distancias para clústering

o para toma de decisiones basadas en aprendizaje automático, e incluso para la propia infraestructura de hardware necesaria para el desarrollo de la Web, tales como enrutamiento, puntos de acceso y distancia a repetidores de señal. Es por esto por lo que resulta de especial utilidad reducir nuestro problema a un problema de grafos, pues este es un campo ampliamente estudiado y desarrollado en el campo de la informática y la computación, por lo que existen una gran diversidad de herramientas, técnicas y utilidades que nos serán de gran ayuda a la hora de resolver el problema que nos atañe.

Desde este punto de vista, la "computación social" se basa en gran medida en el análisis de las redes sociales existentes. Las redes sociales tienen una gran variedad de características que se pueden analizar, siendo su estructura, que normalmente se divide en comunidades, una de las más útiles y provechosas [1]**¡Error! No se encuentra el origen de la referencia.**. Y es precisamente este problema, el problema de la detección de comunidades, el que queremos resolver en cuestión en este Trabajo de Fin de Grado. En términos académicos, las comunidades, conocidas también como clústeres o módulos, son grupos de vértices que tienden a compartir algunas propiedades y/o a cumplir un papel parecido dentro del grafo. La exploración y sondeo de las comunidades dentro de un grafo puede ayudarnos a entender cómo funciona una determinada red [1].

Desde un punto de vista más teórico, y basado en el grado de los nodos, podemos dar una definición más exacta del concepto de comunidad, tal y como se dispone en 0. Siendo $G = (V, E)$, donde V representa el conjunto de nodos (o "Vértices") del grafo, y E representa el conjunto de las aristas del mismo (o "Edges", en inglés). Teniendo en cuenta el grado interno y externo de un nodo i (definido grado interno como el número de aristas que entran al nodo, y grado externo como el número de aristas que salen del nodo), y considerando S como un subgrafo del grafo G al que el nodo i pertenece, entonces podemos afirmar que S es una comunidad en sentido fuerte si para cualquier i perteneciente a S el grado interno de i es mayor que el grado externo de i, y en el sentido débil si el sumatorio del grado interno de cada nodo i de S es mayor que el respectivo sumatorio del grado externo [1]

Desde un punto de vista más general, se pueden entender las comunidades como ciertas regiones de la red que se está tratando donde la conectividad es más fuerte o la interacción es más intensa, o, en una noción más relajada, donde se observa entre sus elementos una cierta característica común. Las comunidades pueden ser explícitas, en el caso de que se

conozca específicamente la pertenencia de los elementos que la componen a un grupo común, o pueden detectarse por algún tipo de discontinuidad en la estructura de la red (conectividad, cohesión, etc.), y suelen producirse, en el caso de las redes sociales reales, por algún parecido entre las personas que las componen.

Ahora que tenemos claros los conceptos sobre grafos, y como una red social puede representarse como un grafo, así como el problema que queremos resolver y su importancia en el análisis de características de redes sociales, cabe preguntarnos cómo resolveremos dicho problema. En el apartado siguiente hablaremos de algunos de los métodos que podemos utilizar para resolver este problema.

2.2 Algoritmos de enjambre y su aplicación a la detección de comunidades.

Si tenemos en cuenta algunos parámetros propios del análisis de redes sociales como son la densidad o la modularidad de una red (estos conceptos se explicarán más adelante, en el apartado de análisis) y estudiamos como varían en función de las distintas comunidades de cada red analizada, resulta intuitivo llegar a la conclusión de que podríamos darle la vuelta al problema: dada una red en la que no conocemos su estructura en comunidades, podemos apoyarnos en las métricas anteriormente mencionadas para buscar una subdivisión en comunidades de la red que maximice esos parámetros [5].

Desde este punto de vista, el problema de la detección de comunidades se convierte en un problema de clústering, o un problema de optimización, con una métrica que mida el valor de cohesión, densidad o modularidad de la red como función objetivo.

Como el coste del problema es NP (Nondeterministic Polinomial time, tiempo polinomial no determinista) por el número exponencial del espacio de soluciones [5], lo más sensato sería aproximarnos a él mediante una solución heurística. Para ello, rápidamente vienen a la mente los algoritmos evolutivos como una solución candidata a nuestro problema de optimización.

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos, se mantiene un conjunto de entidades

que representan posibles soluciones al problema que se quiere resolver, las cuales se mezclan y/o compiten entre sí, evolucionando a lo largo del tiempo hacia soluciones más efectivas. Se basa en la evolución de las especies, pues sólo las más adaptadas (las soluciones más aptas para solucionar el problema en cuestión) sobreviven con el paso de las generaciones [4]. Estos algoritmos constituyen una rama de la inteligencia artificial, y son realmente útiles para resolver problemas donde el espacio de búsqueda es extenso y no lineal, pues son capaces de encontrar soluciones aptas y eficientes en un tiempo razonable, donde otros algoritmos más tradicionales no lo consiguen, por lo que se adaptan perfectamente a nuestro problema.

Para este Trabajo de Fin de Grado, para resolver nuestro problema hemos puesto nuestra atención en un algoritmo bio-inspirado llamado "Particle Swarm Optimization", o "Optimización por Enjambre de Partículas", en adelante, "PSO".

Contrariamente a los algoritmos evolutivos, el optimizador por enjambre de partículas, en su versión canónica, no está basado en la selección, pues típicamente todos los miembros de la población sobreviven desde el principio de la prueba hasta el final, resultando sus iteraciones en una mejora cualitativa de las soluciones a lo largo del tiempo [3]. PSO basa su funcionamiento en el comportamiento de ciertos animales sociales, como cuando los peces nadan formando un cardumen o banco de peces, los pájaros se organizan en bandadas o los insectos se comportan como un enjambre (de esta última similitud viene el hecho de que estos algoritmos bio-inspirados como PSO se conozcan habitualmente como "algoritmos de enjambre").

PSO es un algoritmo bio-inspirado que permite optimizar un problema computacional empleando un grupo de partículas, de forma que cada partícula es una solución candidata al problema de optimización. Las soluciones candidatas se actualizan en base a reglas simples aprendidas por las partículas (función de fitness). *Gracias a su eficacia y su facilidad de implementación, PSO es un algoritmo predominante en el campo de los problemas de optimización* [1]**¡Error! No se encuentra el origen de la referencia.**.

Sin embargo, existe un último escollo que deberemos superar si queremos utilizar este algoritmo para aplicarlo al problema de detección de comunidades que venimos describiendo a lo largo de este punto, y es que la versión canónica de este algoritmo está diseñada para

problemas de optimización en espacios continuos, mientras que en nuestro problema de grafos estamos ante un espacio discreto. Deberemos pues adaptar el algoritmo original de forma que pueda aplicarse a un espacio discreto como es nuestro problema de detección de comunidades en redes sociales.

Para ello, hemos realizado una serie de modificaciones sobre el algoritmo original, las cuales convierten PSO en un algoritmo voraz y discreto, que aprovecha las ventajas de la topología de las redes (grafos), y que describiremos en detalle más adelante.

2.2.1 Particle Swarm Optimization

PSO es un algoritmo de búsqueda estocástico y basado en la población que fue propuesto por primera vez por J. Kennedy y R. Eberhart en 1995, y está inspirado en el comportamiento de ciertos animales sociales que podemos observar en la naturaleza tales como los bancos de peces o las bandadas de pájaros, donde los animales se organizan de forma inteligente para beneficiar sus propósitos gracias a una conducta grupal consensuada.

Su fácil implementación, estructura sencilla y rápida convergencia computacional hacen de PSO una técnica de optimización muy popular para resolver problemas de optimización continuos [1]**¡Error! No se encuentra el origen de la referencia.**.

PSO trabaja con un grupo de individuos, que típicamente son llamados "partículas". En este algoritmo, cada partícula tiene una posición y una velocidad, representados por vectores. El vector de posición típicamente representa una solución candidata al problema de optimización que se está tratando, y el vector de velocidad indica la tendencia a cambiar de posición de la partícula, es decir, de algún modo representa como de cerca o lejos está esa solución de la solución ideal u óptima.

Para buscar la solución óptima, la partícula actualiza su posición y velocidad iterativamente de acuerdo con su propia experiencia y de acuerdo también con la experiencia de otras partículas (nótese que se denomina experiencia de la partícula, o del enjambre, a los datos sensibles obtenidos de la codificación de soluciones previas más óptimas que la posición/solución actual). Si asumimos que el tamaño del enjambre de partículas que vamos

a utilizar es *n*, y tomamos $V_i = \{v_i^1, v_i^2, v_i^3, \ldots, v_i^D\}$ y $X_i = \{x_i^1, x_i^2, x_i^3, \ldots, x_i^D\}$ como los *i*-ésimos vectores de velocidad y posición para una partícula (con *i*=1, 2, …, *n*), respectivamente, tenemos que, en la versión canónica de PSO, una partícula actualiza su velocidad y posición de acuerdo con las siguientes reglas simples [1]:

$$V_i = V_i + c_i r_i (Pbest_i - X_i) + c_2 r_2 (Gbest - X_i) \qquad \textit{(Ecuación 1)}$$

$$X_i = X_i + V_i \qquad \textit{(Ecuación 2)}$$

En la ecuación anterior existen algunos términos que conviene explicar y clarificar:

- *Pbest*: $Pbest_i = \{pbest_i^1, pbest_i^2, pbest_i^3, \ldots, pbest_i^D\}$ representa la mejor posición personal de la partícula *i* del enjambre, es decir, representa cual es, hasta el momento, la solución más cercana a la óptima a la que esa partícula ha llegado.
- *Gbest*: $Gbest_i = \{gbest_i^1, gbest_i^2, gbest_i^3, \ldots, gbest_i^D\}$, parecido al caso anterior, pero en esta ocasión representa la mejor posición global de todo el enjambre (atendiendo a la función a maximizar elegida), es decir, representa la mejor solución global hasta el momento, a la que deberían "seguir" el resto de las partículas para que el enjambre vaya por buen camino en su búsqueda de una solución definitiva y eficiente.
- r_1 y r_2: números aleatorios entre cero y uno.
- c_1 y c_2: coeficientes de aceleración denominados componente cognitiva y componente social.

Numerosos investigadores de este campo han trabajado en mejorar el rendimiento del enjambre en diferentes formas, tales como los descritos en [6] y[7], siendo en [7] donde se introduce por primera vez el concepto de "peso de inercia" en este algoritmo, añadiéndolo a las reglas de actualización de velocidad tal y como se muestra a continuación:

$$V_i = \omega V_i + c_i r_i (Pbest_i - X_i) + c_2 r_2 (Gbest - X_i) \qquad \textit{(Ecuación 3)}$$

Los autores argumentan que un valor de ω relativamente alto favorecería la exploración, mientras que un valor pequeño del mismo favorece la explotación. Las diferencias entre exploración y explotación en el lenguaje que nos atañe se pueden consultar en detalle en [8], y básicamente se trata de la exploración de nuevas soluciones y posibilidades frente a la explotación de principios que sabemos ciertos. Es decir, se trata de buscar un balance en nuestra búsqueda entre la medida en la que se exploran nuevas soluciones y distintas de las

actuales, o la medida en la que al encontrar una buena solución no nos separamos de ella en exceso, explotándola en lugar de buscar nuevos horizontes. Para alcanzar este balance en el algoritmo, en [7] se propone un factor de inercia decreciente linealmente, aunque también se han estudiado complejos métodos de optimización no lineal para este parámetro [9].

3 Diseño

3.1 Adaptando PSO a escenarios discretos

La versión canónica del algoritmo que nos ocupa (PSO, Particle Swarm Optimization), descrita a lo largo de la sección anterior fue diseñada para optimizaciones en escenarios continuos. Sin embargo, las ventajas inherentes a este algoritmo han hecho que en los últimos años algunos académicos hayan intentado llevar este algoritmo a escenarios discretos. El primer intento de esto fue propuesto por los propios J. Kennedy y R. Eberhart en [10], se basó en un esquema de codificación binaria, y fue llamado BPSO (Binary PSO). Aunque para determinados escenarios este algoritmo demostró funcionar bien y satisfacer los problemas que llevaron a su proposición, el esquema de codificación binaria sigue limitando su aplicación a muchos problemas de optimización como el que nos atañe. También existen otros métodos directos basados en técnicas de transformación del espacio tales como las propuestas en [11] ó [12].

Nosotros, en este Trabajo de Fin de Grado, por nuestra parte, optaremos por una solución más simple y efectiva para el problema que nos ocupa (el de la detección de comunidades), basado en matrices y arrays, redefiniendo los operadores necesarios para adaptar las fórmulas de la versión canónica de PSO a un escenario discreto, donde operaremos con matrices, listas de nodos y subgrafos, y que se describirá con mayor detalle en los apartados que siguen a este.

3.2 Detalles del algoritmo propuesto

Siguiendo con lo mencionado en el apartado 3.1, era nuestro propósito para este Trabajo de Fin de Grado el aplicar un algoritmo de enjambre, en este caso el descrito PSO, sobre un problema de grafos, en este caso, el descrito de la detección de comunidades sobre el grafo de Facebook.

Para resolver este problema de optimización mediante la técnica descrita, resulta necesario realizar algunas modificaciones sobre la forma canónica del algoritmo de Particle Swarm

Optimization para adaptarlo a un espacio discreto y de esta manera poder resolver el mencionado problema de grafos.

Las modificaciones propuestas, que se definirán a continuación, tienen como objetivo que el algoritmo haga uso de la topología de la red (de la topología del grafo) para dirigir hacia dónde se mueve el estado de la partícula con cada actualización , es decir, aprovecharemos la topología del grafo para reescribir las normas de actualización del algoritmo canónico vistas en el apartado anterior en (Ecuación 3) y (Ecuación 1) de forma que podamos actualizar la posición y velocidad de las partículas para guiarlas a regiones prometedoras del espacio de soluciones (esto es, hacia soluciones más eficaces).

Para ello, en esta propuesta se introducen nuevos operadores, nuevas formas de inicialización del estado de las partículas y algoritmos de reordenamiento de posiciones para facilitar la convergencia del algoritmo a un espacio discreto.

3.2.1 Explicación general del algoritmo propuesto

El algoritmo propuesto recogerá como parámetros el tamaño del enjambre de partículas, *n*, el número de iteraciones, *gmax*, el valor del peso de inercia, ω y los factores de aprendizaje c_1 y c_2, y dispondrá también como entrada al algoritmo del grafo sobre el que se quieren detectar las comunidades, representado como un grafo de *NetworkX*, cuya implementación se describirá en el apartado de *Desarrollo*, por lo que se podrá disponer también, si fuese necesario, de la matriz de adyacencia del grafo.

La salida del algoritmo estará constituida por la solución al problema, es decir, la división en comunidades, así como el fitness máximo obtenido a lo largo de todas las generaciones del algoritmo y correspondiente a esa partición en comunidades.

Cada partícula entonces será una solución al problema, es decir, será una tentativa de la partición en comunidades del grafo, en forma de array cuyo índice representa los nodos del grafo, y su valor representa la comunidad a la que pertenece. Esto se detallará con mayor precisión más adelante en el presente documento, así como la representación de los vectores de velocidad de cada partícula.

Una vez teniendo en cuenta estos datos de entrada y parámetros variables de los que disponemos, pasaremos a describir de forma general el funcionamiento del algoritmo, el cual describiremos de forma más detallada y exhaustiva en futuros apartados.

- En primer lugar, inicializaremos la población, es decir, estableceremos el estado y las variables que corresponderán a la población inicial. En nuestro caso, para cada partícula del enjambre inicializaremos su vector de posición de forma aleatoria, y su vector de velocidad a 0. Estos vectores tendrán longitud *n*, siendo *n* la dimensionalidad del problema a solucionar. Además, inicializaremos el vector *Pbest* para que sea igual que el vector posición.
- Una vez hecho esto, evaluaremos el fitness de cada partícula de acuerdo con la función de fitness (ver apartado 3.2.2).
- Ahora que tenemos el fitness para cada una de las partículas del enjambre, actualizaremos la partícula *Gbest*, que, tal y como se ha definido en el apartado 2.1.1, se trata de la partícula cuya posición es la más cercana a la solución buscada, es decir, es la partícula con mayor fitness de todo el enjambre. Por ello, *Gbest* cogerá el valor de la partícula con el mayor fitness devuelto tras el punto anterior.
- En este punto comenzará el bucle principal del algoritmo, en el cuál, para cada generación (siempre que el índice del bucle sea menor que *gmax*) haremos lo siguiente (para cada partícula para cada generación):
    - Actualizaremos el estado de la partícula según lo expuesto en el apartado 2.1.1, pero con las modificaciones oportunas que se detallarán en el apartado 3.2.3.
    - Convertiremos la posición resultante en una equivalente que sea mínima en términos computacionales (ver apartado 3.2.5).
    - Evaluaremos el fitness de la partícula en cuestión.
    - Con el dato del fitness, sabremos si debemos actualizar el *Pbest* de la partícula (que es, según lo descrito en 2.1.1, la posición de fitness máximo personal de la partícula). Si el fitness de la partícula en la posición actual es mayor que el fitness de la partícula en la posición marcada por *Pbest*, entonces se asignará a *Pbest* la posición actual de la partícula.

    - Actualizaremos también con este nuevo dato la partícula *Gbest*, que será en máximo fitness global, es decir, será la partícula con mayor fitness devuelto del enjambre.
- Cuando sobrepasemos en el índice del bucle el valor de *gmax*, saldremos del bucle, pararemos el algoritmo y ofreceremos el output, es decir, la mejor solución encontrada, con su división en comunidades propuesta correspondiente y el valor de la función de fitness para su vector de posición.

3.2.2 La función de fitness

En un algoritmo bio-inspirado, como el que estamos tratando en este Trabajo de Fin de Grado, la función de fitness es aquella función objetivo que se pretende maximizar a través de las iteraciones del algoritmo.

En el problema de la detección de comunidades, hemos apuntado con anterioridad que lo más sensato sería utilizar como función de fitness alguna métrica relativa a la estructura de la red que estamos tratando, de forma que podamos detectar la formación de comunidades en base a cambios en la estructura de la red.

Estos cambios en la estructura de la red se pueden medir en base a cambios en los niveles de conectividad, cohesión, modularidad o densidad de la red. Como se ha expuesto en el apartado 2.1, el grado de los nodos puede tener un papel importante en la división en comunidades de la red. También se apuntó en ese apartado que, para detectar comunidades, se trataba de buscar regiones de la red en la que se maximizasen las similitudes entre los nodos que quedan dentro de ella y se minimizasen sus diferencias, al tiempo que buscamos que las diferencias entre los elementos incluidos dentro de una comunidad y los que no lo están sean mayores.

Con esto en mente, hemos definido una función de fitness (cuyos detalles de implementación serán expuestos y discutidos con mayor profundidad en el apartado de *Diseño*) que se basa en las densidades “intra-cluster” e “inter-cluster” de la red. La idea principal de esta función de fitness es que una solución es buena cuando las comunidades que propone están muy cohesionadas en el sentido de que tienen muchas aristas entre nodos incluidos en ellas, y

además están muy desconectadas entre ellas (existen pocas conexiones entre nodos miembros de comunidades diferentes).

La medida de la "densidad intra-cluster" responde a la siguiente fórmula:

$$\delta_{int}(\mathcal{C}) = \frac{\#\text{ internal edges of } \mathcal{C}}{n_c(n_c - 1)/2}.$$

Figura Diseño-1: Densidad intra-cluster

Como vemos, dada una comunidad, la densidad intra-cluster de dicha comunidad se define como el número de aristas internas de la comunidad entre la mitad del producto del número de nodos contenidos en la comunidad por el número de nodos contenidos en la comunidad menos uno, que representa el total de aristas posibles dentro de la comunidad.

Por su parte, la "densidad inter-cluster" sigue la siguiente fórmula:

$$\delta_{ext}(\mathcal{C}) = \frac{\#\text{ inter-cluster edges of } \mathcal{C}}{n_c(n - n_c)}.$$

Figura Diseño-2: Densidad inter-cluster

En la cual, dada una comunidad, el resultado se obtiene dividiendo el número de conexiones entre nodos contenidos en la comunidad y nodos fuera de ella ("inter-cluster edges") entre el producto del número de nodos de la comunidad por la diferencia entre el número de nodos del grafo entre el número de nodos presentes en la comunidad. Este divisor representaría el número máximo posible de conexiones intercomunitarias del grafo en cuestión, representando por tanto esta medida el ratio de conexiones intercomunitarias presentes sobre el número de conexiones intercomunitarias posibles.

Como nuestra función de fitness busca maximizar la relación entre nodos dentro de una comunidad y minimizar esta cohesión entre nodos de distintas comunidades, utilizaremos para conseguir nuestro objetivo la diferencia entre las métricas anteriores descritas, es decir, la densidad intra-cluster menos la densidad inter-cluster:

$$\delta_{int}(\mathcal{C}) - \delta_{ext}(\mathcal{C})$$

Figura Diseño-3: Función de fitness

Con esta función de fitness, es nuestro objetivo hacer que las partículas con posiciones que representen comunidades más diferenciadas se vean beneficiadas a lo largo de las generaciones, de manera que el enjambre "las siga" y las soluciones encontradas mejoren, conduciendo a las partículas hasta regiones prometedoras del espacio de soluciones.

3.2.3 Representación e inicialización de partículas y parámetros

Al hablar del algoritmo PSO hemos hablado de términos como "posición" y "velocidad", y de cómo tendríamos que redefinirlos para adaptarlos a nuestro problema de detección de comunidades en redes sociales. A continuación, se describe como se han modificado y redefinido dichos términos y elementos para adaptarlos a un espacio discreto y poder resolver el problema que nos atañe.

En primer lugar, hemos redefinido la "posición" de la partícula como un vector que representa una partición de una red. Este vector se representará mediante una lista de enteros tal que $X_i = [x_i^1, x_i^2, \ldots, x_i^n]$, con $x_i^j \in [1, n]$ donde el tamaño de la lista es *n*, que además coincidirá con el número de nodos del grafo. En esta lista, cada elemento x_i^j es un identificador que representa la información de la comunidad en la que se sitúa dicho nodo, y donde el índice de la lista, es decir, $j \in [1, 2, \ldots, n]$ representa el nodo per se.

Esta codificación de "posición discreta" es sencilla y fácil de decodificar, y a su vez reducirá el coste computacional del algoritmo, especialmente para grafos de gran escala, ya que la dimensión de la función de fitness será la misma que el número de nodos de la red.

A continuación se muestra un grafo de ejemplo para ilustrar las explicaciones:

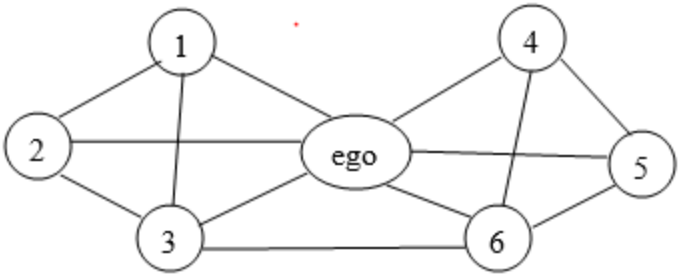

Figura Diseño-4: Grafo de ejemplo

En el caso del grafo que vemos sobre estas líneas, se trata de un grafo en forma de ego-network, tal y como los vamos a tratar en la práctica, pues en esta forma se presenta el dataset de Facebook utilizado. Un grafo en forma de red ego se representa como las conexiones de un nodo con todos sus vecinos (denominados *alter*), y las conexiones de esos vecinos entre ellos. En el ejemplo vemos como la red ego del nodo "ego" se compone de los nodos 1, 2, 3, 4, 5 y 6, los cuales están todos ellos conectados al nodo central.

Para las posibles soluciones, es decir, para las posiciones de las partículas, no tendremos en cuenta el ego a la hora de formar comunidades, ya que podemos notar a simple vista como en una red ego, el propio "ego" constituye el centro, pues todos los nodos están conectados con él, por lo que podría clasificarse en cualquier comunidad y no habría manera alguna de saber si es más correcto que forme parte de una u otra comunidad.

Tendríamos por tanto para este grafo de ejemplo, unos vectores de posición para cada partícula que serían listas de tamaño 6, y que se parecerían a lo siguiente:

[1, 1, 2, 3, 3, 2]

Esta sobre estas líneas podría ser una primera solución tentativa, una posición de una partícula, o incluso la posición de una partícula de primera generación generada al azar. Si atendemos a lo explicado más arriba, esta posición nos indica la división en comunidades propuesta por la partícula en cuestión. En este caso, nos estaría diciendo que los nodos 1 y 2 (con índices comenzando en 1) están en una comunidad, los nodos 4 y 5 en otra, y los nodos 3 y 6 en otra, es decir, sería el equivalente a dividir el grafo de ejemplo en las siguientes comunidades:

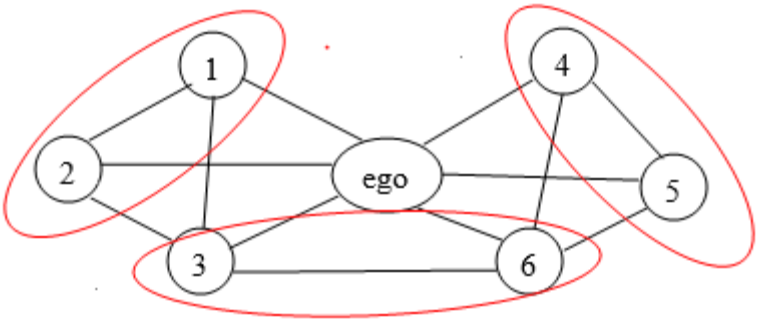

Figura Diseño-5: Posible solución

La solución es coherente, pero no del todo acertada, puesto que, a simple vista, siendo el grafo de un tamaño tan reducido, puede verse que la solución más clara y razonable para dividirlo en comunidades es la siguiente:

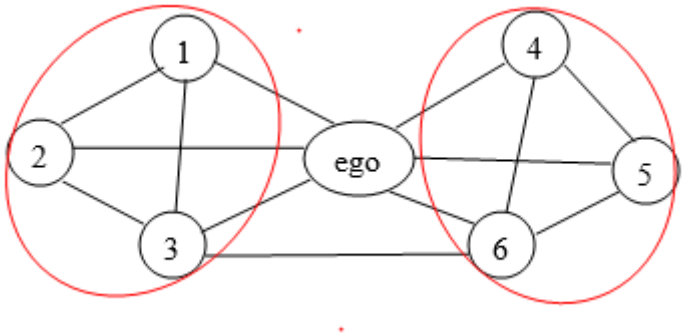

Figura Diseño-6: Solución óptima

Por lo que, si este grafo de ejemplo fuese el grafo sobre el que aplicásemos el algoritmo desarrollado, esta es la solución a la que querríamos llegar aplicando nuestra función de fitness y nuestras funciones de actualización de estado de las partículas.

Esta solución codificada mediante el sistema de "posiciones discretas" o "covers" que estamos utilizando para representar la división en comunidades, quedaría como sigue:

[1, 1, 1, 2, 2, 2]

Además, hemos redefinido también la "velocidad" de una partícula de forma similar a lo que hemos hecho con la posición, pero con algunas diferencias que comentaremos a continuación.

El vector velocidad de la partícula, que representará la tendencia de la partícula a moverse y la dirección hacia la que se moverá, se representará mediante una lista de enteros tal que $V_i = [v_i^1, v_i^2, \ldots, v_i^n]$, con $v_i^j \in \{0, 1\}$

Si el elemento v_i^j del vector velocidad es 1, significa que el correspondiente elemento x_i^j del vector velocidad será cambiado, mientras que, si es 0, mantendrá su estado original. Si un determinado valor del vector posición tiene que cambiar porque así lo indica la correspondiente posición del vector velocidad, lo hará según las distintas reglas de actualización de estado de las partículas, que se definirán a continuación, en el apartado 3.2.4.

En cuanto a la inicialización, los vectores de posición de las partículas se inicializarán a valores aleatorios, es decir, se rellenarán las *n* posiciones con enteros aleatorios entre 1 y *n*. Los vectores de velocidad se inicializarán a cero, es decir, tendrán ceros en todas las posiciones de la lista que los representa. Los vectores *pbest*, es decir, los que representan la mejor posición de la partícula, serán inicializados de la misma forma que los vectores de posición, y el vector *gbest* será la mejor posición de la población original.

3.2.4 Reglas de actualización del estado de las partículas

Tal y como hemos expuesto en el apartado anterior, los vectores de posición y velocidad de una partícula han sido redefinidos para adaptarse a un espacio discreto y por tanto poder resolver nuestro problema de grafos. Por tanto, se hace necesario de la misma manera redefinir las reglas de actualización de las partículas, de forma que dichas partículas cambien de posición según una velocidad determinada en función de un fitness y lleguen a producir soluciones mejores a nuestro problema con cada iteración.

Para ello ha sido necesario definir algunos operadores nuevos, que en nuestro código se convertirán en funciones, tal y como se detallará más adelante en el apartado de *Desarrollo*.

Como el lector seguramente recordará, en el apartado 2.1.1 definimos las reglas de actualización de la versión canónica de PSO como sigue:

$$V_i = \omega V_i + c_i r_i (Pbest_i - X_i) + c_2 r_2 (Gbest - X_i) \quad \textit{(Ecuación 3)}$$

$$X_i = X_i + V_i \quad \textit{(Ecuación 2)}$$

Para poder adaptarlas a nuestros vectores de posición y velocidad modificados, será necesario definir al menos las/los siguientes funciones/operadores:

Substracción de posiciones:

En el espacio discreto y teniendo en cuenta los vectores que hemos diseñado, no podemos simplemente restar aritméticamente un vector al otro, ya que en este caso un vector de posición menos otro vector de posición deberá dar como resultado una velocidad, la cual por definición estará formada por un vector en el que sus elementos sólo pueden ser 0 ó 1. Por tanto, la solución propuesta para esta función es la siguiente:

Dadas dos posiciones P1 = [p_1^1, p_1^2, …, p_1^n] y P2 = [p_2^1, p_2^2, …, p_2^n], la resta de dichas posiciones devolverá un vector velocidad V = [v_1, v_2, …, v_n], donde cada elemento v_i se definirá como 0 si $p_1^i = p_2^i$, y como 1 si $p_1^i \neq p_2^i$, es decir, la posición correspondiente del vector de velocidad resultante de la resta de dos vectores posición será cero si dicho elemento es igual en P1 y en P2, y 1 si es diferente.

Este operador se ha definido así por varias razones. La primera de ellas es que, en la teoría de la inteligencia de enjambres, una partícula ajusta su velocidad aprendiendo de sus vecinos, lo que nos empuja a pensar que el proceso de aprendizaje en esta fase es similar una comparación entre posiciones, ya que la partícula observa su posición con respecto de la de sus vecinos y actualiza su velocidad en consecuencia [1].

Por otro lado, desde el punto de vista de la teoría de grafos, las dos posiciones representan dos estructuras en comunidades diferentes de la red, por lo que el operador de substracción definido refleja las diferencias entre las dos estructuras de red.

Multiplicación de un coeficiente por una velocidad:

Este es probablemente el operador más simple de los que hemos tenido que definir, pues únicamente se limita a, dado un vector de posición y un coeficiente, multiplica cada posición del vector de velocidad por el coeficiente dado, realizando una multiplicación aritmética básica. A continuación, se puede ver un ejemplo que ilustra el comportamiento descrito:

Vector velocidad V = [1, 0, 0, 1, 0, 1] Coeficiente c1 = 1.72

Resultado de la multiplicación: c1*V = [1.72, 0, 0, 1.72, 0, 1.72]

Suma de velocidades:

Este operador trata de sumar dos velocidades, devolviendo como resultado otra velocidad fruto de la suma de ambas. No obstante, debe asegurarse de que se sigue manteniendo la condición de que los vectores de velocidad están codificados en binario (sus elementos solo pueden tomar valores de 0 ó 1).

Para ello, el operador se define como sigue:

Dados dos vectores de velocidad V1 = [v_1^1, v_1^2, …, v_1^n] y V2 = [v_2^1, v_2^2, …, v_2^n], el resultado será del mismo modo otro vector de velocidad V3 = [v_3^1, v_3^2, …, v_3^n] tal que cada elemento v_3^i se define como 1 si la suma de v_1^i y v_2^i es mayor o igual que 1, y cero en caso contrario (menor que 1).

Así se consigue mantener velocidades codificadas en binario de forma que es más fácil para las posiciones operar con ellas.

Posición por velocidad:

Este operador es un componente clave del algoritmo que estamos desarrollando, ya que, en última instancia, será el encargado de llevar a la partícula a una región prometedora del espacio "multiplicando" la posición por la velocidad, de forma que se genere una nueva posición. Es decir, este será el operador encargado de "arrastrar" a una partícula hasta una nueva posición donde pueda encontrar un mayor fitness. Este operador se ha codificado de la siguiente manera:

Dada una posición P1 = [p_1^1, p_1^2, …, p_1^n] y una velocidad V1 = [v_1^1, v_1^2, …, v_1^n], el resultado será una nueva posición P2 = [p_2^1, p_2^2, …, p_2^n], de forma que cada elemento p_2^i será:

- El elemento de la posición antigua, $p_2^i = p_1^i$, en caso de que v_1^i sea 0.
- Si la velocidad v_1^i es 1, entonces p_2^i es el máximo de las diferencias de fitness entre la posición antigua y cada uno de sus vecinos. Para ello, se genera una lista con los identificadores (es decir, p_1^j) de los vecinos (en el grafo) del nodo i, y para cada

elemento de la lista (esto es, para cada vecino) se cambia en la posición P1 el identificador del nodo i en cuestión por el identificador del vecino, y se comprueba el fitness de la nueva posición, al cual se le restará el fitness de la posición antigua y se guardará esta diferencia. A continuación, se elegirá la máxima diferencia de fitness, y el vecino que la haya conseguido pasa a formar parte de la nueva posición.

De esta forma, las partículas actualizan su posición eligiendo el identificador del vecino que puede generar un mayor fitness.

Como el concepto puede resultar un poco confuso, a continuación, se describe un ejemplo del modo de actuar de este operador.

Supongamos que tenemos el siguiente grafo:

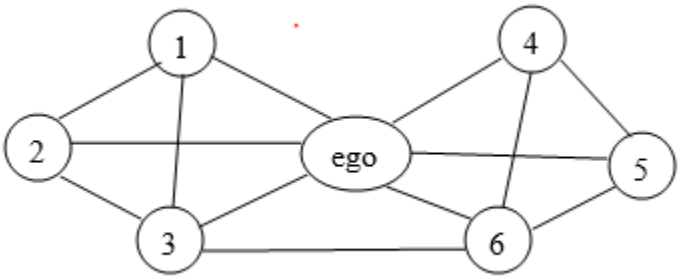

Figura Diseño-7: Grafo de ejemplo (II)

Con un vector posición X = [1, 1, 2, 3, 3, 2] y un vector de velocidad V = [0, 1, 0, 0, 0, 0]

Esto nos indicaría que deberemos actualizar el elemento 2 (con índices desde 1) del vector de posición X. El operador, por tanto, generará, ignorando el ego, una lista de los identificadores de los vecinos del nodo 2. Los vecinos del nodo 2 son (excluyendo el ego), los nodos 1 y 3, cuyos identificadores en el vector de posición X son 1 (para el 1) y 2 (para el 3).

Se generará por tanto una lista con los identificadores [1, 2], y se sustituirá cada uno de ellos por el identificador del nodo antiguo, es decir, se generará, primero, la posición [1, 1, 2, 3, 3, 2], y después la posición [1, 2, 2, 3, 3, 2], y se calculará, para cada una de ellas, la diferencia de fitness con la posición X original, eligiéndose aquella que maximice esa diferencia. Si suponemos que el fitness para la posición [1, 2, 2, 3, 3, 2] es mayor que el fitness que tenía X, la diferencia fitness *([1, 2, 2, 3, 3, 2])-fitness(X)* es mayor que la

diferencia fitness *([1, 1, 2, 3, 3, 2])-fitness(X)* (que en este caso será cero, pues son la misma posición), a la salida del operador, la nueva posición X_{new} será [1, 2, 2, 3, 3, 2].

3.2.5 Reducción a posiciones equivalentes.

En el algoritmo desarrollado, como las posiciones iniciales de la población inicial de partículas se generan al azar dentro del espacio de búsqueda, y además intervienen operaciones aritméticas y operaciones entre listas que pueden hacer crecer los números que representan la comunidad en la que esta cada nodo, es posible que lleguemos a situaciones donde el identificador de la comunidad a la que pertenece cada nodo esté en valores por encima de los necesarios para determinar el número de comunidades máximo, es decir, podríamos estar ante una situación en la que nuestro vector de posición sea como sigue:

$$X = [3, 4, 3, 3, 4, 4]$$

Lo cual indicaría la existencia de dos comunidades, una para los nodos 1, 3 y 4, y otra para los nodos 2, 5 y 6. La motivación de esta función que vamos a explicar a continuación no es otra que hallar un vector de posición equivalente, pero con el mínimo número de identificadores, y que los mismos tengan un valor que sea el mínimo posible. De esta forma podremos detectar posiciones equivalentes y evitar un extra de gasto computacional innecesario. Es decir, siguiendo con el ejemplo anterior, lo que buscamos es una nueva posición, equivalente a la anterior, pero cuyos identificadores de comunidad empiecen en el 1, el mínimo valor posible. En este caso sería:

$$X = [1, 2, 1, 1, 2, 2]$$

De esta manera, si diseñamos esta operación y asumimos que las dos posiciones anteriores son en realidad la misma (puesto que representan la misma solución candidata al problema de detección de comunidades), a la hora de realizar una substracción de posiciones equivalentes en la operación definida en el apartado 3.2.4, el resultado sería un vector de velocidad que es cero. Lo cual resultaría en no calcular una nueva posición al actualizar el estado, pues no ganaremos nada con el hecho de movernos a una posición que es la misma en la que ya estamos, con lo que disponer de esta función e incluirla en el algoritmo resulta en un ahorro considerable del tiempo de computación.

4 Desarrollo

4.1 Detalles de la implementación realizada

Como se ha comentado en secciones anteriores del presente documento, el objetivo de este Trabajo de Fin de Grado es desarrollar una versión discreta y voraz del algoritmo de enjambre PSO (Particle Swarm Optimization), de forma que lo podamos aplicar al problema de la detección de comunidades en la estructura de una red social (en nuestro caso Facebook) que se comporta como un grafo, pudiendo extrapolar y aplicar por tanto conocimientos propios de la teoría de grafos a nuestra solución.

A lo largo de esta sección, se hablará del proceso de desarrollo del algoritmo en cuestión, según el diseño planteado en el apartado 3, y se describirán detalles como el lenguaje de programación en el que ha sido desarrollado, las técnicas que se han utilizado para llevar a cabo el algoritmo, y cómo se han codificado las funciones de actualización de partículas descritas en el apartado 3, así como las estructuras de datos requeridas para su realización.

4.1.1 Lenguaje de programación

Para la implementación del algoritmo principal desarrollado durante este Trabajo de Fin de Grado, se ha elegido el lenguaje de programación Python. Este lenguaje ha demostrado adaptarse muy bien al tipo de problema que nos ocupa por, entre otras muchas, las siguientes razones:

- Existen librerías externas de gran calidad que nos facilitan enormemente el trabajo y manejo de ciertas estructuras de datos, tales como los grafos, que son la pieza fundamental del algoritmo desarrollado. Se ha usado la librería NetworkX [13] para el manejo de grafos y Numpy [14] para el tratamiento eficiente de arrays y matrices multidimensionales, entre otras.
- Es un lenguaje que permite una gran facilidad a la hora de tratar listas y arrays donde se puede guardar la información relevante para el problema que nos ocupa.
- Gran parte de los algoritmos bio-inspirados tienen desarrollos en Python. Es un lenguaje cuya popularidad en el campo del aprendizaje automático y la inteligencia artificial está en ascenso.
- Es un lenguaje eficiente para manejar grandes cantidades de datos.

- Al ser un lenguaje de alto nivel y de gran potencia, podemos codificar algoritmos complejos de manera más sencilla, útil y eficiente.
- Es un lenguaje que permite aprovechar las ventajas tanto de la programación orientada a objetos como de la programación funcional.

4.1.2 Ficheros de código desarrollados

Para llevar a cabo la implementación de este algoritmo basado en PSO que resuelva problemas de grafos (detección de comunidades en redes sociales), hemos implementado dos ficheros de código Python: `Particula.py` y `Enjambre.py`.

`Particula.py` representa una partícula, con sus características tales como su posición, velocidad y vector *Pbest*, así como las funciones necesarias para los operadores entre posiciones y velocidades descritos en el apartado 3.2.4.

`Enjambre.py` representa el enjambre de partículas, y será el fichero que contiene el bucle principal del algoritmo. Este fichero será el que habrá que ejecutar para lanzar el algoritmo sobre el grafo deseado. Contiene el enjambre de partículas, los métodos para cargar el grafo a partir de los ficheros necesarios para describir la red ego, el bucle principal del algoritmo y algunas funciones auxiliares.

4.1.3 Carga de datos e inicialización

Como hemos expuesto en la introducción, la red social con la que vamos a trabajar para resolver nuestro problema de detección de comunidades mediante un algoritmo de enjambre basado en PSO es una partición anonimizada de Facebook en forma de 10 ego-networks. Una ego-network, o una red ego, consiste en un nodo central, normalmente denominado "ego", y los nodos a los que este "ego" está directamente conectado (nodos a distancia 1), los cuales son habitualmente conocidos como "alters", así como los vínculos, si los hubiese, entre los nodos "alters".

Por ejemplo, el grafo que veíamos antes en los ejemplos del apartado de diseño es un ego-network:

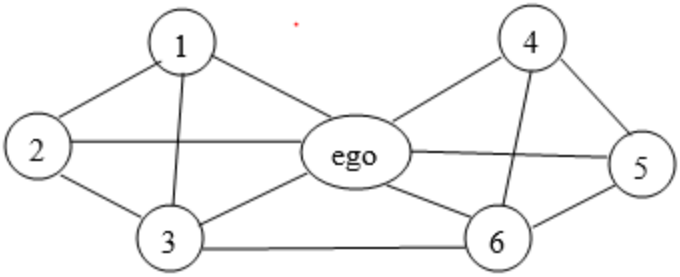

Figura Desarrollo-8: Grafo de ejemplo (III)

El dataset elegido para nuestro Trabajo Fin de Grado, que puede consultarse en 0, está formado por 10 redes ego, correspondientes a 10 usuarios distintos con todos sus amigos y las conexiones entre ellos, debidamente anonimizado. Para cada red ego, se dispone de 5 ficheros distintos, cuyos nombres son de la forma "[id del nodo].[tipo de fichero]", los cuales se describen a continuación:

- [nodeId].edges: Fichero que contiene las aristas presentes en la red ego para el nodo *nodeId*. Estas aristas son no dirigidas para el grafo de Facebook (serían dirigidas, por ejemplo, para datasets de Twitter o Google Plus, donde existen relaciones tanto unidireccionales como bidireccionales), por lo que el fichero únicamente contiene una serie de filas en cada una figuran el nodo origen y el nodo destino de la relación, siendo equivalente, por ejemplo, la relación "2 3" con la relación "3 2". Las conexiones entre el nodo ego y cada alter no aparecen, pero se asume que cada uno de los nodos que aparece en el fichero está a su vez conectado con el nodo ego (puesto que, si no lo estuviese, no aparecería en este fichero).
- [nodeId].feat: Contiene las características de cada uno de los nodos que aparecen en el fichero [nodeId].edges. Estas características se expresan en codificación binaria (1 si la cumple, 0 si no), existiendo otro fichero que relaciona la posición de cada característica con su nombre, para poder realizar la "traducción". Estas características se corresponden con la información que dichos usuarios han indicado dentro de la red social.
- [nodeId].featnames: Este es el fichero que contiene la "traducción" que se mencionaba en el punto anterior. Este fichero ha sido anonimizado para los usuarios de Facebook, puesto que el nombre de las características podría revelar datos personales.
- [nodeId].egofeat: Contiene las características del nodo ego (el propio nodo *nodeId*).

- [nodeId].circles: Contiene los círculos, o comunidades, presentes en la red. Cada línea de este fichero está compuesta por el nombre del círculo y una lista de ids que representan los nodos que están incluidos en dicha comunidad.

Para nuestro trabajo, utilizaremos fundamentalmente los ficheros [nodeId].feat y [nodeId].edges.

El fichero [nodeId].feat lo utilizaremos para saber cuáles son los nodos conectados al ego, de modo que se nos presenten ordenados y sepamos también cuál será el tamaño del grafo al crear nuestro grafo de NetworkX. Como no vamos a hacer un análisis de división en comunidades orientado a las características de los usuarios que las componen, sino orientado al número de aristas entre casa nodo de cada comunidad con respecto del número de aristas externas a dicha comunidad, no nos interesará la parte de las características en sí del fichero feat, sino únicamente la primera posición, donde aparece el id de cada uno de los nodos que forman la red ego que estamos tratando.

El fichero [nodeId].edges lo utilizaremos para añadir a nuestro grafo de NetworkX las aristas entre los nodos "alters".

Para cargar el grafo se ha diseñado una función loadGraph(self, feat, edges) la cual recibirá los siguientes parámetros de entrada:

- Feat: referencia al fichero que contiene las características del nodo, es decir, referencia al fichero [nodeId].feat descrito anteriormente.
- Edges: referencia al fichero que contiene las aristas entre "alters", es decir, referencia al fichero [nodeId].edges descrito anteriormente.

La función en primer lugar cargará el nodo ego en la primera posición del grafo para, a continuación, ir añadiendo un nodo por cada línea del fichero feat, cogiendo únicamente la primera posición de cada línea e introduciéndola en una lista. A continuación, añadirá todos los elementos de la lista al grafo mediante la función *add_nodes_from(nodeList)* de NetworkX [13].

De la misma forma, a continuación abrirá el fichero edges, y para cada línea leerá la primera y la segunda posición, correspondientes al nodo origen y el nodo destino de la arista, respectivamente, y lo introducirá al grafo mediante la función *add_edge(nodeOrigin, nodeDest)* de NetworkX [13].

Una vez hecho esto, devolverá el grafo creado. Esta función se define en el fichero *Enjambre.py*.

4.1.4 Definiendo una partícula

Como Python puede considerarse un lenguaje orientado a objetos, podríamos entender una partícula como un objeto de tipo Partícula, con sus atributos y sus métodos. En el fichero *Partícula.py* se define la clase Partícula, que representa esto mismo.

Así pues, si atendemos a las definiciones expuestas en el apartado 3.1.4, cada partícula tendrá los siguientes atributos:

- Posición: vector definido como una lista, que contendrá los identificadores de los nodos, representando la información sobre en qué comunidad está cada nodo.
- Velocidad: vector definido como una lista, que contendrá información sobre si el índice correspondiente del vector posición debe ser modificado o no. Codificado en binario (1 si debe modificarse, 0 si no). Representa la tendencia de la partícula a moverse a otra posición.
- Pbest: vector, representado como una lista, que guarda la mejor posición personal para esa partícula, es decir, la posición que consigue un mayor fitness.
- Graph: cada partícula contiene una instancia del grafo sobre el que se trabaja, para facilitar el diseño y la implementación de las funciones que se describirán a continuación.
- graphParse y graphParseRevese: diccionarios que relacionan el índice del nodo con su nombre. En el caso de graphParseReverse, tiene invertidas las claves con los valores para facilitar la búsqueda de claves por valores.

A continuación, se describe el código desarrollado para implementar las funciones y operadores descritas a lo largo del apartado 3.2.4.

Substracción de posiciones:

Para implementar esta funcionalidad, se ha codificado la función *positionSubstraction(self, position1, position2)* en el fichero *Particula.py*, dentro de la clase *Particula*. La función recibe como argumentos los siguientes parámetros:

- position1: un vector posición, representado como una lista.
- position2: otro vector posición, representado también como una lista

La función comprobará primero que las posiciones que ha recibido sean listas de la misma longitud y no sean vacías. En caso de que se cumpla que tienen longitudes diferentes y/o una de las dos está vacía y/o las dos están vacías, se informará del error y se parará la ejecución devolviendo un error.

Una vez verificado que las posiciones son válidas, la función creará un nuevo vector de velocidad inicialmente a cero, y recorrerá la longitud de los vectores posición, comprobando si para cada índice de la lista, el elemento en position1 es igual al elemento en position2. Si son iguales introducirá un 0 en ese índice del vector velocidad, y en caso contrario introducirá un 1.

Al acabar el bucle devolverá el vector velocidad resultante.

Multiplicación de un coeficiente por una velocidad:

Para implementar esta funcionalidad, se ha codificado la función *coefficientMultiplyV(self, coefficient, velocity)* en el fichero *Particula.py*, dentro de la clase *Particula*. Esta función recibe como argumento los siguientes elementos:

- coefficient: el coeficiente por el que se va a multiplicar la velocidad. Normalmente es un número decimal de doble precisión o coma flotante.
- velocity: un vector de velocidad, representado como una lista.

La función comprobará que el vector de posición suministrado no es nulo, y una vez hecho esto creará un nuevo vector (lista) en el que irá guardando los nuevos valores resultantes de multiplicar el coeficiente por cada uno de los elementos de la lista, obteniendo así una nueva lista cuyos elementos son los de la primera lista, pero multiplicados por el valor de *coefficient*.

Suma de velocidades:

Para implementar esta funcionalidad, se ha codificado la función *velocityAddition(self, velocity1, velocity2)* en el fichero *Particula.py*, dentro de la clase *Particula*. Esta función recibe como argumento los siguientes elementos:

- velocity1: un vector velocidad, representado como una lista.
- velocity2: otro vector velocidad, representado también como una lista

La función comprobará primero que las velocidades que ha recibido sean listas de la misma longitud y no sean vacías. En caso de que se cumpla que tienen longitudes diferentes y/o una de las dos está vacía y/o las dos están vacías, se informará del error y se parará la ejecución devolviendo un error.

Como, según el algoritmo desarrollado, la suma de dos velocidades debe resultar en una nueva velocidad, una vez comprobado esto, se creará una nueva velocidad inicialmente a cero, y se iniciará un bucle de tamaño el tamaño de los vectores velocidad. En el bucle se recorrerán ambas listas, y se sumarán los valores con el mismo índice de ambas listas, comprobando si dicho valor es mayor o igual que 1, en cuyo caso se introducirá un 1 en ese índice del vector velocidad resultado, o es menor que uno, en cuyo caso se introducirá un cero.

Al salir del bucle se devolverá la velocidad resultante.

Posición por velocidad:

Para implementar esta funcionalidad, se ha codificado la función *positionXVelocity(self, position, velocity)* en el fichero *Particula.py*, dentro de la clase *Particula*. Esta función recibe como argumento los siguientes elementos:

- position: un vector posición, codificado en forma de lista.
- velocity: un vector velocidad, codificado también en forma de lista.

Esta función codifica la funcionalidad estrella del algoritmo, pues será la encargada de actualizar la posición de la partícula con respecto de la velocidad, y para ello ha de tener en cuenta el fitness.

La función comprobará primero que las listas (velocidad y posición) que ha recibido sean listas de la misma longitud y no sean vacías. En caso de que se cumpla que tienen longitudes diferentes y/o una de las dos está vacía y/o las dos están vacías, se informará del error y se parará la ejecución devolviendo un error.

Una vez verificado esto comenzará el código que implementa la funcionalidad descrita para esta función en 3.2.4. Para ello, la función iniciará un bucle desde cero hasta el tamaño de las listas, y en primer lugar comprobará si ese elemento de la lista que constituye el vector velocidad es cero o uno. En caso de que sea cero, asignará a ese índice de la posición resultante el valor de la posición antigua (es decir, la recibida como argumento).

En caso de que la velocidad sea 1 para ese índice, se creará una lista con los vecinos del nodo que representa dicho índice, mediante la función *neighbors(node)* de NetworkX [13]. A continuación, iteramos sobre la lista de vecinos y, para cada vecino, modificamos la posición originar cambiando el índice que se está tratando por el valor del identificador del vecino correspondiente, es decir, si i es la variable de control del bucle principal (sobre los elementos de la lista constitutivos del vector posición), j es la variable de control del bucle sobre la lista de vecinos y l es la lista de vecinos del nodo i+1 (debido al desfase originado por contar los nodos desde 1 y las posiciones de la lista desde 0), asignaríamos:

positionNew[i] = position[l[j]]

Una vez hecho esto, calcularíamos el fitness de la nueva posición y lo compararíamos con el de la antigua haciendo una resta, es decir, dado fitNew = fit(positionNew) y fitOld = fit(position), haríamos difFit = fitNew-fitOld. Si esa diferencia resulta ser más grande que la diferencia máxima almacenada, la diferencia máxima pasa a ser esta nueva, guardándose también la posición con la que se consigue el fitness que origina una diferencia tan grande con respecto a fitOld.

Terminado este bucle interno sobre la lista de vecinos, se elige cuál ha sido el vecino que ha originado la diferencia de fitness más alto, y se asigna de forma definitiva el identificador de ese vecino al índice de la posición nueva en la que estemos.

Finalizado el bucle externo, se devolverá la nueva posición creada.

Función de fitness:

Para codificar la función de fitness se han codificado las siguientes funciones en el fichero *Particula.py* dentro de la clase *Particula*:

- *intracluster(self, community)*: implementa la métrica intracluster tal y como se define en el apartado 3.2.4, utilizando las funciones oportunas de la librería NetworkX para obtener el número de aristas y nodos necesario en cada caso.
- *intercluster(self, community)*: implementa la métrica intercluster tal y como se define en el apartado 3.2.4, utilizando las funciones oportunas de la librería NetworkX para obtener el número de aristas y nodos necesario en cada caso.
- *computeMetric(self, communities)*: recibe como argumento una lista de comunidades *communities*, que en realidad es una lista de listas, pues cada comunidad es una lista de los nodos que la componen. Para cada comunidad de la lista de comunidades acumula en un resultado parcial la resta de la métrica intracluster para esa comunidad menos la métrica intrercluster para esa misma comunidad. Finalmente divide ese resultado parcial acumulado entre el número de comunidades, que será la longitud de la lista de comunidades.
- *coverToCompute*: esta función convierte la posición de la partícula, que está en forma de "cover" tal y como se explica en el ejemplo del apartado 3.2.3 en una lista de comunidades. Es decir, convierte, por ejemplo, la posición [1, 2, 1, 2, 3, 3] en la lista de listas [[1, 3], [2, 4], [5,6]], de forma que pueda ser recibida consecuentemente por la función *computeMetric*, que necesita dicha lista de listas.

Actualización de estado de la partícula:

Para la codificación de esta funcionalidad se ha definido la función *updateStatus(self, gbest)* en el fichero *Particula.py* dentro de la clase *Particula*. La misión de esta función es, recibiendo *gbest* como argumento, que es la partícula con el máximo fitness general del enjambre, aplicar las reglas de actualización de una partícula del algoritmo PSO, tal y como vienen definidas en (Ecuación 3) y (Ecuación 2) en el apartado 2.1.1, pero sustituyendo los operadores aritméticos por nuestras nuevas funciones, redefiniendo así las reglas de actualización de partículas de PSO canónico, que quedarían como sigue:

$$V_i = velocityAddition(\ coefficientMultiplyV(\omega, V_i\ ), velocityAddition(\ coefficientMultiplyV(c_1 \times r_1, positionSubstraction(Pbest_i, X_i)),\ coefficientMultiplyV(c_2 \times r_2, positionSubstraction(\text{Gbest}, X_i)))$$ *(Ecuación 4)*

$$X_i = positionXvelocity(X_i, V_i)$$ *(Ecuación 5)*

4.1.5 Enjambre y bucle principal

En el fichero *Enjambre.py* se ha definido la clase *Enjambre*, donde además de definirse la función que cargará el grafo a partir de los ficheros de aristas y características de la red ego, se ha definido el código necesario para implementar lo descrito en el apartado 3.2.1 en lo que constituye el bucle principal del algoritmo.
En primer lugar, se definirán los valores de los parámetros de *popSize* (tamaño de la población, o tamaño del enjambre) y *gmax* (número de generaciones, número de veces que se ejecuta el algoritmo en busca de soluciones con mayor fitness).

Acto seguido se inicializará la población, para lo cual se creará un array de elementos de tipo Particula con posición inicial aleatoria (generada asignando los elementos que constituyen el vector mediante llamadas a la función *random.randint(inf, sup)* acotado entre los valores 1 y el número de nodos del grafo. Para las velocidades, se crea una lista con 0 en todos sus índices.

Después se evalúa el valor de fitness de cada partícula de la población inicial y se almacenan los resultados en un diccionario, lo cual nos ayudará en el futuro a recorrer los fitness máximos para actualizar la partícula *gbest* en caso de ser necesario. El diccionario tendrá como clave la partícula y como valor su fitness.

Una vez tenemos este primer diccionario de partículas con sus fitness para la población inicial, asignamos *gbest* como la clave con el máximo valor de dicho diccionario.

En este momento comenzamos el bucle de iteraciones mediante un bucle *while* desde 0 hasta *gmax*. Dentro del bucle, para cada partícula llamamos a la función *updateStatus* previamente descrita, y a continuación realizamos la conversión a una partícula mínima equivalente, según lo descrito en el apartado 3.2.5, para lo cual se ha implementado la función *reorder_permutation* en la clase *Particula*.

Una vez hecho esto calcularemos el fitness para las nuevas posiciones llamando a nuestra función *fit*, que no hace otra cosa que llamar a su vez a la función *computeMetric* definida anteriormente. Con los resultados, actualizaremos el diccionario de partícula/fitness y el

vector pBest de cada partícula en caso de ser necesario (en caso de haberse obtenido un fitness mayor para las nuevas posiciones con respecto a las antiguas).

Al salir del bucle interno de cada partícula en el enjambre, actualizaremos de nuevo *gbest* como la partícula con mayor fitness del enjambre y sumaremos uno a la variable del bucle de iteraciones. Cuando esta variable sea mayor que *gmax*, pararemos el bucle y el algoritmo, mostraremos el output y finalizaremos la ejecución.

4.2 Problemas encontrados durante la fase de desarrollo y soluciones implementadas

Algunos de los principales problemas encontrados durante la realización del desarrollo descrito durante los apartados 3 y 4 estuvieron relacionados con el tiempo de ejecución del algoritmo, puesto que para valores altos de *gmax* y de *popSize* los tiempos de ejecución se iban a valores lejanos a los razonables para resolver el problema de una manera ágil y eficiente.

Para solucionar este problema se decidió hacer una revisión general del código del algoritmo para detectar puntos donde fuese posible optimizar el tiempo de ejecución, reformulando secciones de código, bucles y condiciones para agilizar al máximo la ejecución del programa.

En primer lugar, decidimos cambiar muchas de nuestras operaciones con listas nativas de Python por arrays de la librería Numpy [15], los cuales tratan los datos y las operaciones matemáticas y de conjuntos de una forma más rápida, puesto que están optimizados para ello.

Otro foco de las grandes pérdidas de tiempo que estábamos experimentando resultó ser la función *positionXvelocity*. Una revisión exhaustiva de la misma nos llevó a darnos cuenta de que se estaba calculando el fitness para la posición antigua en todas las iteraciones del bucle interno de la lista de vecinos del nodo (ver apartado 4.1.4 para clarificaciones sobre el código de la función), cuando esto podía hacerse calculando este fitness una sola vez y almacenándolo en memoria para luego ir comparando cada nuevo fitness con este *fitOld* almacenado, con el consiguiente ahorro en coste computacional que esto conlleva.

También detectamos algún que otro problema en la actualización de las partículas, ya que estas no parecían ir tan claramente como se había planeado hacia un buen fitness, sino que parecían estancarse en valores de fitness bastante pobre.

De nuevo, un análisis exhaustivo del código reveló que existía un error en la función *positionXvelocity* por el cual se asignaba a la nueva posición el índice del vecino (es decir, el propio vecino en sí) en lugar de su identificador (la información sobre la comunidad a la que pertenece).

Otro problema recurrente tuvo que ver con la forma de ignorar el ego en el fitness, ya que, si se incluye en las comunidades al ego, como es equidistante de todas ellas, podría llevar a la confusión al algoritmo y perjudicar su toma de decisiones. Esto quedó solventado con la incorporación de nuevas métricas para el fitness, y definiendo listas para las posiciones de dimensión número de nodos del grafo menos uno, que empezasen desde la posición 1, y asignando al ego en la carga del grafo el valor 0.

5 Pruebas y resultados

5.1 Descripción de las pruebas realizadas

A fin de constatar que el algoritmo desarrollado funciona correctamente y resuelve el problema para el cual fue diseñado, se han establecido una serie de pruebas que determinarán la validez del algoritmo desarrollado en su aplicación al problema de la detección de comunidades en grafos de redes sociales.

Existen en el algoritmo desarrollado dos parámetros fundamentales, que están codificados para ser aceptados como argumentos del programa por la línea de comandos y que influyen directamente en los resultados del algoritmo, y son los siguientes:

- *gmax*: número de iteraciones del algoritmo
- *popSize*: tamaño del enjambre de partículas

En función de como se ajusten los parámetros pueden variar en gran medida las comunidades que se encuentran, el fitness máximo y el tiempo de ejecución. Como veremos más adelante, en la siguiente sección, donde llevaremos a cabo una serie de pruebas par ajustar estos parámetros, la tendencia parece indicar que a más número de iteraciones y mayor tamaño del enjambre el fitness tiende a subir. No obstante, se hace necesario buscar un compromiso óptimo entre el valor de estos parámetros, el fitness obtenido y el tiempo de ejecución, para mantener el tiempo de ejecución del algoritmo dentro de unos límites razonables.

En primer lugar, antes lanzarnos a realizar pruebas más grandes, exhaustivas y costosas, para verificar que el algoritmo desarrollado funciona según lo esperado (es decir, comprueba de validación del algoritmo) y devuelve como output una división en comunidades de la red que resulta coherente con el grafo analizado en cuestión, se plantea una prueba en la que el grafo analizado por el algoritmo sería el grafo que venimos viendo como ejemplo a lo largo de las diversas secciones del presente documento. Es decir, teniendo el grafo siguiente:

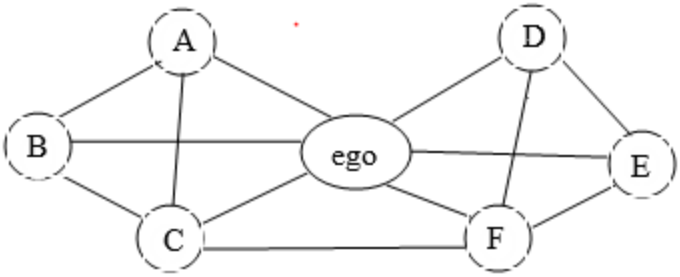

Figura Pruebas-9: Grafo de ejemplo (Letras)

Como se trata de un grafo sencillo y de pequeño tamaño, podemos analizar su estructura a simple vista, y determinar a mano alzada cuál sería su estructura en comunidades. Con una observación rápida del grafo, es sencillo determinar que, pese al enlace existente entre los nodos C y F, en este grafo pueden observarse principalmente dos comunidades, una formada por los nodos A, B y C, y la otra formada por los nodos D, E, F; por lo que ese será el resultado que esperamos que devuelva el algoritmo desarrollado al pasarle este grafo de prueba como argumento para verificar que está funcionando correctamente.

Ejecutamos por tanto el algoritmo sobre el grafo mostrado en *Figura Pruebas-1*, con gmax=25 y popSize=25, obteniendo el siguiente output:

Output:
Best fitness: 0.634920634921
Communities:
Community 1: ['A', 'B', 'C']
Community 2: ['D', 'E', 'F']
Time elapsed: 0.476s

Como vemos, obtenemos las comunidades esperadas, con un fitness relativamente alto (basado, como se ha explicado anteriormente, en la diferencia entre la densidad intracluster y la densidad intercluster), de 0.6349, lo cual es un buen dato, y en un tiempo bastante corto, de menos de medio segundo. Podemos concluir, por tanto, que el algoritmo, al menos sobre el grafo de prueba, funciona según lo esperado, detectando las comunidades presentes en la red con un acierto bastante aceptable y similar al que deduciríamos nosotros en nuestro raciocinio humano al observar la estructura del grafo.

Una vez hecho esto, y verificado que el algoritmo funciona bien, y realiza la función para la cuál ha sido concebido, podemos pasar a las pruebas más exhaustivas, pruebas que realizaremos sobre los datasets de Facebook descritos en la sección 4.1.3, cuyos resultados se mostrarán con detalle a continuación en el apartado siguiente.

Las pruebas consistirán en, para cada uno de los "egos" obtenidos del dataset de Facebook de Snap [15], ejecutaremos el algoritmo sobre él con diferentes valores de *gmax* y *popSize*, y observaremos lo que va ocurriendo con el fitness, las comunidades obtenidas y el tiempo de ejecución.

Tendremos además la posibilidad de comparar las comunidades devueltas por el algoritmo contra el fichero *circles* de ese ego, viendo en cuanto se aproxima lo detectado por el algoritmo desarrollado en función de la métrica de fitness utilizada a las comunidades reales existentes en la red (las comunidades reales para cada uno de los egos probados se pueden encontrar en el Anexo 1).

5.2 Resultados obtenidos y discusión

5.2.1 Resultados obtenidos

A continuación, se muestran los resultados obtenidos para el dataset de Facebook, presentados por "ego". Se ha resaltado en verde el fitness máximo general.

Ego 3980 (ficheros 3980.edges y 3980.feat), 59 nodos:

popSize	**gmax**	**Mejor fitness**	**Tiempo de ejecución (s)**
5	5	0.122151173	1.02974820
5	10	0.241801764	2.04631400
5	25	0.407698059	4.34074091

5	50	0.356707318	20.1691780
5	**100**	**0.681474047**	**24.7614450**
10	10	0.262422673	5.23980402
10	25	0.269284038	12.3473789
10	50	0.19853312	20.4683310
10	100	0.549008606	57.666254
25	10	0.285691873	10.6102960
25	25	0.480913283	33.8016798
25	100	0.431342500	221.100114
50	10	0.268085848	20.8322751
50	25	0.385339921	73.6384840
50	50	0.384978619	158.046250
50	100	0.515253303	378.283698

Tabla 1: Algunas ejecuciones del algoritmo sobre el ego 3980

Ego 698 (ficheros 698.edges y 698.feat), 66 nodos:

pop Size	gmax	Mejor fitness	Tiempo de ejecución (s)
10	10	0.247728293	13.4409451
10	25	0.344120336	42.166433
10	50	0.292741634	130.540481
10	100	0.259536472	226.271917
25	10	0.209752971	26.3338198
25	25	0.174647745	87.1848120
25	**100**	**0.418732240**	**553.219309**
50	10	0.184420002	60.1061348
50	25	0.281445490	242.050807
50	50	0.371538297	639.903060
50	100	0.354271390	1248.36981

Tabla 2: Algunas ejecuciones del algoritmo sobre el ego 698

Ego 414 (ficheros 414.edges y 414.feat), 159 nodos:

popSize	**gmax**	**Mejor fitness**	**Tiempo de ejecución (s)**
10	10	0.052774817	264.982053
10	**25**	**0.272326080**	**1435.81882**

Tabla 3: Algunas ejecuciones del algoritmo sobre el ego 414

5.2.2 Discusión sobre los resultados obtenidos

Si observamos con cierto detenimiento los resultados anteriores podemos observar cómo, si nos atenemos al fitness, parece que al ejecutar el algoritmo con un número de generaciones (*gmax*) bajo y un tamaño de la población (*popSize*) también bajo (situando ambos valores por debajo de 25), vemos como el valor del fitness se ve perjudicado, obteniendo en la mayoría de los casos valores de fitness inferiores a 0.25, lo cual era de esperar, pues este algoritmo de enjambre basado en PSO se basa en una actualización generacional de las partículas, y mientras más generaciones haya, más tiempo de mejorar tendrán las dichas partículas, que, gracias a las reglas de actualización de sus estados, tienden a generar mejores fitness y por ende mejores resultados que en la anterior generación. Podría suceder, no obstante, que, llegado a un determinado número de generaciones, el algoritmo se estancase y su fitness dejase de subir, debido entre otras cosas al factor de convergencia. Es decir, debido a que las partículas han sido guiadas por las funciones de actualización hacia una zona poco prometedora demasiado pronto, y como es característica propia de los algoritmos de enjambre que las partículas tiendan a seguir a otras (a las de mayor fitness) podrían acabar todas en una región demasiado cercana con un fitness no óptimo, tal y como se puede observar, por ejemplo, para el ego 698 en el paso de 50 a 100 generaciones con tamaño de la población 50.

Por otro lado, el tamaño del enjambre también influye normalmente en una mejora del fitness, ya que cuantos más individuos haya, más posibilidades hay de que uno de ellos

alcance un mayor fitness. No obstante, en las pruebas realizadas se observa frecuentemente un efecto de saturación en el enjambre, puesto que en ocasiones al aumentar el tamaño del enjambre para un mismo número de generaciones vemos un impacto negativo en el fitness. Esto puede suceder debido a que un enjambre más grande necesitaría más generaciones para que todas sus partículas acabasen por seguir a la partícula de mayor fitness. Es decir, con un tamaño alto del número de partículas, las partículas cambian de región del espacio de soluciones con más lentitud. Este fenómeno lo podemos observar, por ejemplo, en la tabla del ego 3980, para los casos de *popSize* 25 y *popSize* 50, con *gmax* 25 para ambos.

Vemos como el fitness máximo se ha alcanzado para el ego 3980 para la ejecución con *popSize*=5 y *gmax*=100, con un fitness máximo de 0.6815, y con un tiempo de ejecución de 24.76 segundos. Esto nos lleva a pensar que, sobre los datasets probados, el algoritmo desarrollado reacciona mejor (en cuanto a valor de fitness) a un aumento del número de generaciones que a un aumento del número de partículas en el enjambre.

En cuanto al tiempo de ejecución, vemos como el número de iteraciones *gmax* tiene un impacto directo en el tiempo de ejecución del algoritmo, lo cual es lógico dado que este parámetro indicará el número de veces que se ejecutará el algoritmo principal.

De forma parecida impacta también el número de partículas del enjambre, *popSize*, puesto que cuantas más partículas existan en el enjambre, se tienen que calcular más fitness y actualizar más partículas para cada iteración.

Si realizásemos una comparación entre las comunidades propuestas por el algoritmo desarrollado y las reales, seguramente veríamos como rara vez coinciden más de dos o tres comunidades, incluso aunque el fitness sea bastante alto. A esto podríamos encontrarle explicación en que el algoritmo desarrollado se basa para diferenciar las comunidades en una función de fitness determinada. En el caso de el algoritmo aquí descrito, dictamina que un conjunto de nodos “se parece más” a una comunidad que otro si existe un mayor número de enlaces entre los nodos de ese conjunto que entre nodos de ese conjunto y nodos exteriores al mismo. En el mundo real, sin embargo, las relaciones de amistad en una red social como puede ser Facebook, normalmente no se organizan en torno a métricas estrictas, sino que influyen otros factores como el grado de amistad entre las personas en el mundo real, o el tipo de uso que da la persona a la red social. Por esta razón existen variaciones entre la estructura que indica el algoritmo desarrollado y la real recogida de datos verídicos.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En este Trabajo de Fin de Grado se ha tratado de demostrar que es posible la aplicación de algoritmos de enjambre, en concreto, Particle Swarm Optimization, que ha sido diseñado para espacios continuos, a un problema discreto como es el problema de la detección de comunidades en redes sociales, el cual es, básicamente, un problema de grafos.

El problema de la detección de comunidades en redes sociales tiene especial interés, puesto que, en el mundo actual, y en los últimos años, las redes sociales se están convirtiendo en un nuevo referente en cuanto a información y comunicación. A través de ellas se difunden cantidades ingentes de datos en forma de noticias, relaciones entre usuarios y características de los usuarios que conectan, y a través de ellas se mantienen informados y se comunican miles de millones de personas, por lo que el análisis de datos de redes sociales se ha convertido en un campo muy provechoso para los investigadores y expertos en computación y tratamiento de información con muy diversos objetivos. Es un campo en el que convergen gran cantidad de estudios de diversas ramas del conocimiento, desde las ciencias sociales hasta la informática.

Analizando la estructura de un grafo, o de una red social, podemos llegar a entender cómo funciona, como se han formado sus enlaces y en base a qué su estructura y morfología evolucionan, e incluso podemos llegar a predecir futuros comportamientos, flujos de información y otros fenómenos que se produzcan en la red. Si tenemos una forma de detectar las comunidades de una red, podremos establecer similitudes entre los usuarios que la componen, y viceversa.

Por otra parte, en los últimos años también están ganando popularidad los algoritmos evolutivos, genéticos y de enjambre, ya que son capaces de resolver problemas computacionalmente complejos en un tiempo razonable, ya que no tienen que explorar todo el espacio de búsqueda (tarea que en ocasiones es sencillamente inabordable), sino que se mueven en base a la optimización de una función objetivo, llamada función de fitness. De entre estos nuevos algoritmos se ha elegido PSO debido a que ha atraído mucho interés en los últimos años, y ha demostrado ser una técnica de optimización metaheurística excelente

para problemas de optimización continuos. Sin embargo, el hecho de que PSO haya sido diseñado para dominios continuos limita su operativa en dominios discretos como el que nos ocupa. Otra pega importante de PSO es que sufre de la llamada "maldición de la dimensionalidad", lo cual también lo limita en la aplicación en problemas de optimización de gran escala.

En este Trabajo de Fin de Grado se ha diseñado una implementación alternativa sobre PSO canónico de forma que pueda aplicarse a dominios discretos (como es el caso de nuestro problema de detección de comunidades) y se reduzca en la manera de lo posible la "maldición de la dimensionalidad". Se ha diseñado un algoritmo en el que se han adaptado las estructuras de datos y las reglas de actualización de estado de las partículas con respecto a PSO canónico de manera que aprovechase al máximo la topología de la red y se adaptase correctamente a la resolución del problema de detección de comunidades.

Las pruebas realizadas tanto sobre el grafo de prueba como sobre los grafos de los distintos egos de Facebook demuestran que el algoritmo tiene potencial para resolver el problema para el que fue propuesto, y que con un ligero ajuste de parámetros en función de la topología y número de nodos del grafo sobre el que se busquen las comunidades, se puede llegar a obtener un fitness bastante óptimo y una división en comunidades bastante precisa. No obstante, se ve claramente en las pruebas realizadas que para grafos con una cantidad grande de nodos el tiempo de ejecución del algoritmo se ve gravemente perjudicado, lo cual es sin duda herencia de los problemas de PSO a la hora de tratar problemas de optimización de gran escala, con un coste computacional bastante alto.

6.2 Trabajo futuro

Tal y como se expresa en el apartado anterior, el algoritmo desarrollado, si bien cumple el objetivo para el cual fue diseñado, sigue teniendo alguna que otra pega al enfrentarse a grafos con un número de nodos muy grande, en los cuales el tiempo de ejecución es alto.

En futuros trabajos se trabajará en la optimización del algoritmo presentado, enfocándolo a la resolución de problemas computacionales complejos sobre dominios complejos. Es decir, se realizará un análisis más profundo acerca de cómo re-adaptar el algoritmo diseñado para que sea capaz de hacer frente problemas de tipo LSGO (Large-Scale Global Optimization,

en castellano, Optimización Global a Gran Escala), de forma que podamos mejorar lo ya conseguido en este Trabajo de Fin de Grado que es la adaptación del algoritmo PSO a dominios discretos mediante la creación de un algoritmo basado en PSO que resuelve el problema de la detección de comunidades en redes sociales.

Por otro lado, cabe la posibilidad de comparar el rendimiento de este algoritmo frente a otros algoritmos de similares características, como pueden ser algoritmos de colonias de hormigas, o algoritmos genéticos. Y además, analizar la calidad de las soluciones comparándolas con el groundtruth del dataset, que en este TFG no se ha podido realizar por restricciones de tiempo.

Referencias

[1] Quing Cai, M. Gong, L. Ma, S. Ruan, F. Yuan, L. Jiao, "Greedy discrete particle swarm optimization for large-scale social network clustering", International Research Center for Intelligent Perception and Computation, Xidian University, China.

[2] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, D. Parisi, "Defining and identifying communities in networks", Proc. Nat. Acad. Sci. USA 101 (9) (2004) 2658–2663

[3] Kennedy J. (2011) "Particle Swarm Optimization". Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA

[4] (2011) "Evolutionary Algorithms". Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA.

[5] Equipo docente de la asignatura "Busqueda y Minería de Información" de la UAM, "Análisis de redes sociales", EPS-UAM.

[6] S.-T. Hsieh, T.-Y. Sun, C.-C. Liu, S.-J. Tsai, Efficient population utilization strategy for particle swarm optimizer, IEEE Trans. Syst. Man Cybern. 39 (2) (2009) 444–456.

[7] Y. Shi, R. Eberhart, A modified particle swarm optimizer, in: Proceedings of 1998 IEEE Congress on Evolutionary Computation, 1998, pp. 69–73

[8] James G. March, "Exploration and Exploitation in Organizational Learning", Standford University.

[9] Y. Shi, R. Eberhart, Fuzzy adaptive particle swarm optimization, in: Proceedings of 2001 IEEE Congress on Evolutionary Computation, vol. 1, 2001, pp. 101–106.

[10] J. Kennedy, R. Eberhart, A discrete binary version of the particle swarm algorithm, in: Proceedings of 1997 IEEE International Conference on Systems, Man, and Cybernetics, vol. 5, 1997, pp. 4104–4108.

[11] A. Salman, I. Ahmad, S. Al-Madani, Particle swarm optimization for task assignment problem, Microprocess. Microsyst. 26 (8) (2002) 363–371.

[12] D.Y. Sha, C.Y. Hsu, A hybrid particle swarm optimization for job shop scheduling.

[13] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008.

[14] Travis E, Oliphant. A guide to NumPy, USA: Trelgol Publishing, (2006).

[15] J.Leskovec, Snap Facebook Dataset, https://snap.stanford.edu/data/egonetsFacebook.html

Glosario

PSO	Particle Swarm Optimization
LSGO	Large Scale Global Optimization
Ego-Network	Red de nodos a distancia 1 del nodo central o “ego”
Fitness	Resultado de la función de fitness de un algoritmo evolutivo/genético/de enjambre